



# Runtime Code Polymorphism as a Protection against Physical Attacks

Damien Couroussé, Bruno Robisson, Thierno Barry, P Jaillon, O Potin

## ► To cite this version:

Damien Couroussé, Bruno Robisson, Thierno Barry, P Jaillon, O Potin. Runtime Code Polymorphism as a Protection against Physical Attacks. Workshop on Cryptographic Hardware and Embedded Systems, Sep 2015, Saint-Malo, France. <<http://www.chesworkshop.org/ches2015/>>. <emse-01232662>

**HAL Id: emse-01232662**

**<https://hal-emse.ccsd.cnrs.fr/emse-01232662>**

Submitted on 23 Nov 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

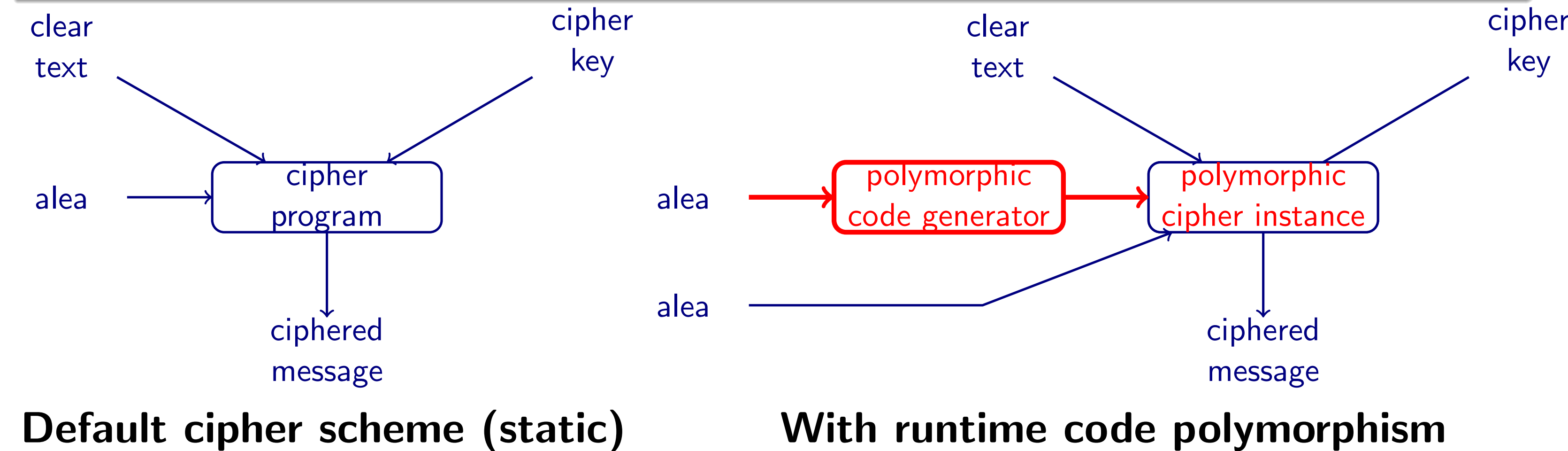
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



## Core Idea: Runtime Code Polymorphism

### Definition

Regularly **changing the behaviour** of a (secured) component, **at runtime**, while maintaining **unchanged** its **functional properties**



### Definition

- Regularly **changing the behaviour** of a (secured) component, **at runtime**, while maintaining **unchanged** its *functional properties*.

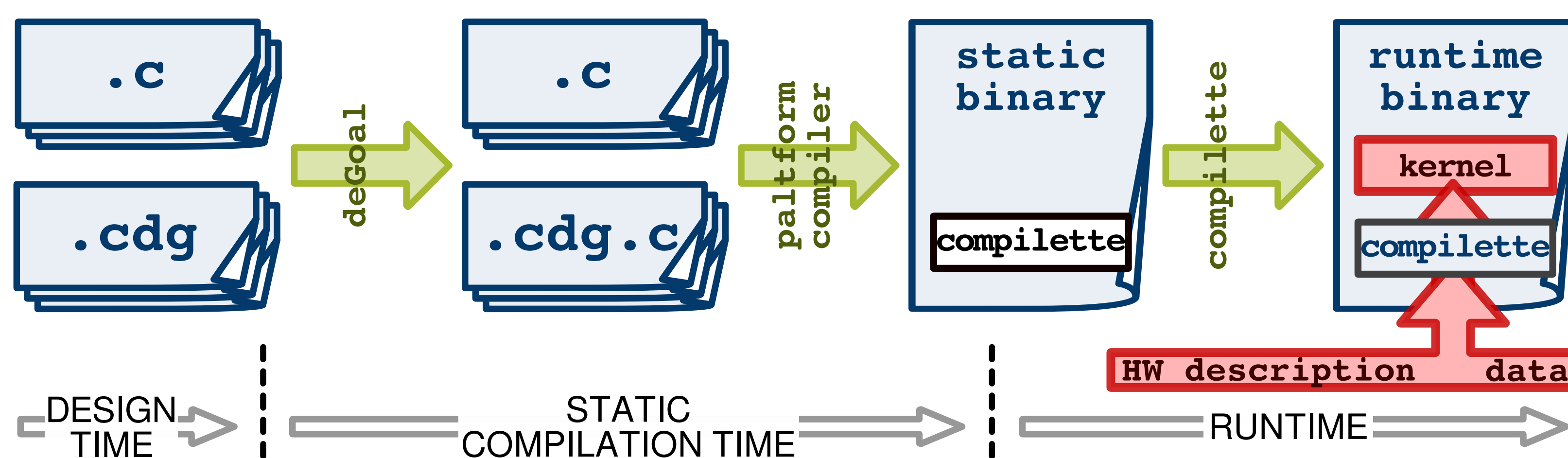
### What for?

- Protection against reverse engineering of SW
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval  $\omega \geq 1$ )
- Protection against physical attacks
  - polymorphism changes the **spatial** and **temporal** properties of the secured code: side channel & fault attacks
  - combine with usual SW protections against focused attacks

### How?

- deGoal: runtime code generation for embedded systems
  - fast code generation
  - tiny memory footprint: proof of concept on TI's MSP430 (512 bytes of RAM)

## Complettes & deGoal in a Nutshell



### Aim

- Modify kernel's binary instructions
- according to the input data
- whenever needed at runtime

### The deGoal framework builds complettes

A compilette is:

- an *ad hoc* code generator that targets *one* kernel
- aimed to be invoked at runtime

## Polymorphic Code Generation

### deGoal runtime capabilities

Performed *in this order*:

- register selection
- instruction selection
- instruction scheduling

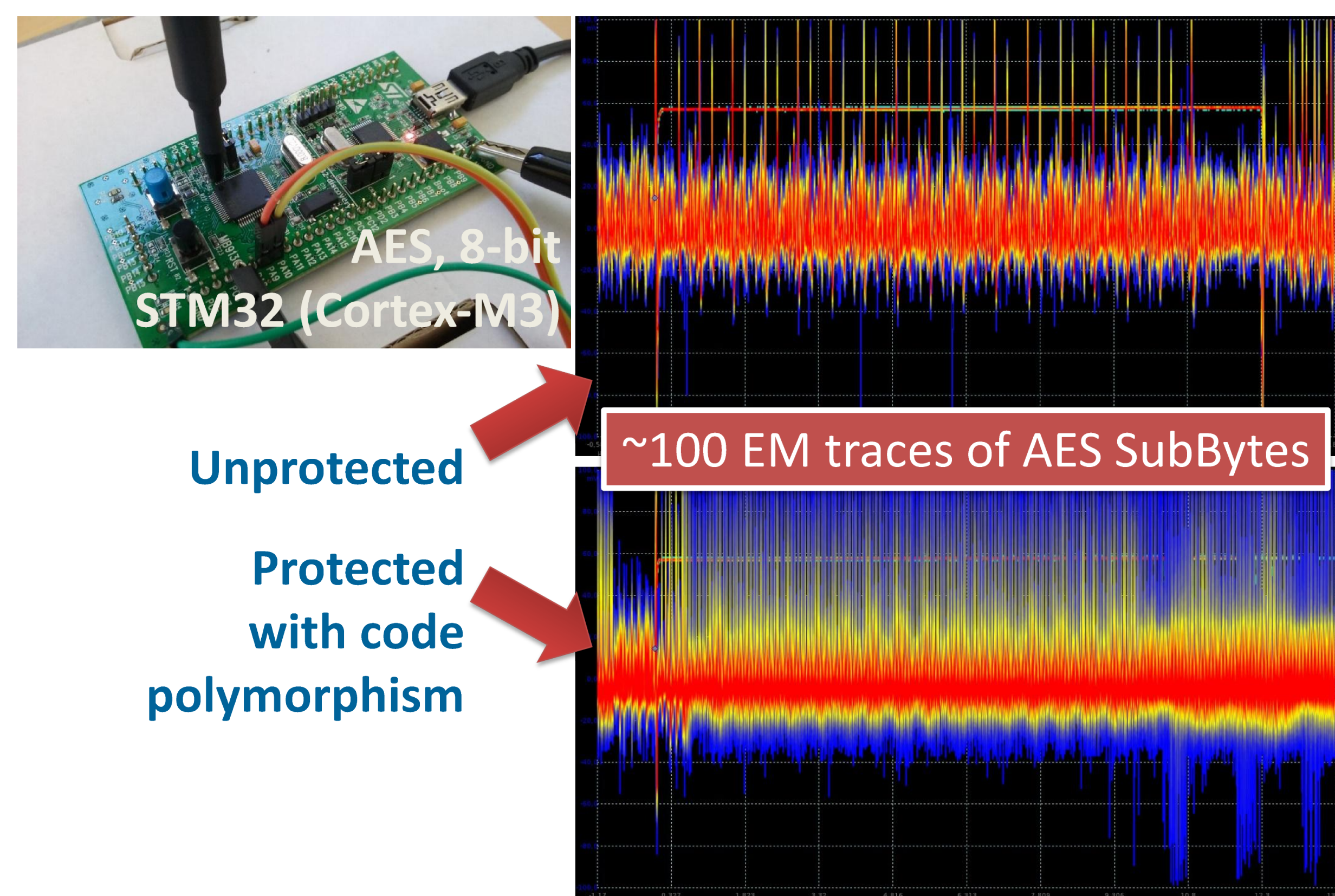
Adaptation to achieve runtime code polymorphism:

- Portability to very small processors and secure elements
  - Limited memory consumption
  - Fast runtime code generation
- Ability to combine with hardware countermeasures
- Introduce alea during runtime code generation [1,2,3]
- Polymorphism:
  - random mapping to physical registers [1]
  - use of semantic equivalences [2]
  - instruction scheduling [3]
  - insertion of dummy operations [3]

## Example: polymorphic AES

Polymorphic implementation of the SubBytes function:

```
void gen_subBytes( cdg_insn_t* code
                  , uint8_t* sbbox_addr
                  , uint8_t* state_addr)
{
    #[
        Begin code Prelude
        Type uint32 int 32
        Alloc uint32 state, sbbox, i, x, y
        mv state, #(state_addr)
        mv sbbox, #(sbbox_addr)
        mv i, #(0)
        loop:
            lb x, @(state+i) // x := state[i]
            lb y, @(sbbox+x) // y := sbbox[x]
            sb @(state+i), y // state[i] := y
            add i, i, #(1)
            bneq loop, i, #(16)
        rtn
    End
    ]#;
}
```



### Execution times (in cycles), over 1000 runs:

	min	max	average
reference	6385	6385	6385
code generator	5671	12910	9345
polymorphic instance	7185	9745	8303

### Impact of the code generation interval $\omega$ :

$\omega$	$k$	%
1	2.76	53.0%
5	1.59	18.4%
20	1.37	2.1%
100	1.31	1.1%

$k$ : overhead vs. reference implementation  
%: percentage contribution of runtime code generation to the performance overhead

### References

Overview of our approach for runtime code generation with complettes:  
H.-P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, and R. Gauguey, "deGoal a Tool to Embed Dynamic Code Generators into Applications," in *Compiler Construction*, 2014, vol. 8409.

Runtime code generation for micro-controllers with less than 1kB RAM:  
C. Aracil and D. Couroussé, "Software Acceleration of Floating-Point Multiplication using Runtime Code Generation," in *Proceedings of the 4th International Conference on Energy Aware Computing*, 2013.

Instruction scheduling for VLIW processors:  
D. Couroussé, V. Lomüller, and H.-P. Charles, *Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform*. Springer Verlag, 2013, pp. 103–124.